Hashed RSA

```python
def computeGCD(x, y):
    while(y):
        x, y = y, x % y
    return x
```

computeGCD()

As the name suggests, the function takes x and y, two numbers as input. It computes and returns their GCD to the calling function.

```python
def encrypt_hash(hash_message, private_key):
    int_message = (int(hash_message, 2) % private_key[1])
    # print(int_message)
    y = ((int_message ** private_key[0]) % private_key[1])

    return y

    # print(int_message)
```

Encrypt_hash()

The function takes message as input, and gives out the encrypted text for the hash function which hashes the encrypted text.

```python
def decrypt_hash(enc_hash, public_key):
    # int_message = int(message, 2)
    return ((enc_hash ** public_key[0]) % public_key[1])
```

```python
# e needs to be co prime with N and Phi_N
def get_encryption_key(N, phi_N):
    possible_e = []
    for i in range(1, N+1):
        if (1 < i) and (i < phi_N):
            gcd = computeGCD(i, N)
            gcd_phi = computeGCD(i, phi_N)
            if (gcd == 1) and (gcd_phi == 1):
                # return i
                possible_e.append(i)
    if len(possible_e) > 1:
        return possible_e[random.randint(1, len(possible_e)-1)]
    else:
        return possible_e[0]
```

Get_encryption_key():

Takes N, and Phi_N as input as parameters. Computes e such that e is co prime with N and Phi_N. Returns a random number from the list of possible e values.

```python
# choose d such that d*e mod phi_N = 1
def get_decryption_key(e, phi_N):
    possible_d = []
    for i in range(e * 25):
        if (e * i) % phi_N == 1:
            possible_d.append(i)
    if len(possible_d) > 1:
        return possible_d[random.randint(1, len(possible_d)-1)]
    else:
        return possible_d[0]
    # return possible_d[random.randint(1, len(possible_d) - 1)]
```

Get_decryption_key()

Takes e and phi_n as input, the function computes d which will be the private key. D is chosen such that d*e mod phi_N = 1. A random d from the possible values of d is returned.

```python
def text_to_digits(message):
    pool = string.ascii_letters + string.punctuation + " "
    # print(pool)
    M = []
    for i in message:
        M.append(pool.index(i))
    return M
```

Text_to_digits()

The function takes message as input and converts each word into an integer of ascii values. Returns a list of such integers.

```python
def digits_to_text(message_digest):
    pool = string.ascii_letters + string.punctuation + " "
    msg = ''
    for i in message_digest:
        msg += pool[i]
    return msg
```

Digits_to_text()

Performs the opposite computation to Text_to_digits(), takes a list of integers as input and computes the string from it.

```python
def encrypt(M, public_key):
    return [(i ** public_key[0]) % public_key[1] for i in M]
```

Encrypt()

Takes message and public key as input and computes the encrypted message by using M^e mod N

```python
def decrypt(CT, private_key):
    return [((i ** private_key[0]) % private_key[1]) for i in CT]
```

Decrypt()

Takes cipher text and private key as input parameters. Computes original digits by using private key, d, M^d mod N.

```python
def message_to_binary(message_lis):
    binary_message = ""
    for i in message_lis:
        binary_message += bin(i).replace('0b', '').zfill(32)
    # print(len(binary_message))
    return binary_message
```

Message_to_binary()

Converts a given message list of integers of ASCII values to binary.

```python
def message_to_binary8bit(message_lis):
    binary_message = ""
    for i in message_lis:
        binary_message += bin(i).replace('0b', '').zfill(8)
    # print(len(binary_message))
    return binary_message
```

Message_to_binary8bit()

Does the same thing as message_to_binary() but restricts the binary to only 8 bits.

```python
def pad_rsa(fixed_bytes, binary_r, null_byte, binary_message):
    return fixed_bytes + binary_r + null_byte + binary_message
```

Pad_rsa()

Converts rsa to padded rsa by padding fixed_bytes, random pad, null bytes to the original message.

```
def remove_pad_rsa(cipher_text):
    message_lis = []
    n = len(cipher_text)//8
    for i in range(n):
        message_lis.append(int(cipher_text[i*8:i*8+8], 2))

    # print(message_lis)
    count = 0
    for i in message_lis:
        count += 1
        # print(count)
        if i == 0:
            break

    message_lis = message_lis[count:]
    encrypted_binary = message_to_binary8bit(message_lis)
    # print(encrypted_binary)

    message_lis = []
    n = len(encrypted_binary)//32
    # print(len(encrypted_binary))
    for i in range(n):
        # print(i)
        message_lis.append(int(encrypted_binary[i*32:i*32+32], 2))

    # print(message_lis)
    return message_lis
```

Remove_pad_rsa()

Takes cipher_text as input, the first step in decryption is to remove the pad added during the encryption, this removal is done by looking for the first NULL byte to know that’s the beginning of the message.

```python
p = 41
q = 59

# Public key N,e
N = p * q
phi_N = (p - 1) * (q - 1)
# print(N)
# print(phi_N)
e = get_encryption_key(N, phi_N)
# print(e)
d = get_decryption_key(e, phi_N)
# print(d)

while d == e:
    d = get_decryption_key(e, phi_N)

public_key = [e, N]
private_key = [d, N]
```

The code snippet here is responsible for choosing p,q and calculates N and phi_N. Both private and public keys are calculated in d and e respectively and we get the complete keys as public_key and private_key.

```python
null_byte = "00000000"

# print(prg.g_calc("011111001000111101001011000100011111100010011011"))
fixed_bytes = "0101001111011010"
binary_r = "0101001111011010011111001000111101001011000100011111100010011011"
# binary_r = message_to_binary(r)
# print(len(binary_r))
# message length should be less than 1014 bytes
message = input("Enter the message to be encrypted: ")

parsed_message = text_to_digits(message)
cipher_text = encrypt(parsed_message, public_key)
binary_cipher = message_to_binary(cipher_text)
cipher_to_send = pad_rsa(fixed_bytes, binary_r, null_byte, binary_cipher)
```

The snippet shows the encryption process, the bytes are padded essential for padded RSA.

Message is taken as input from the user, conversion to digits, digits are then encrypted which are then converted to binary so as to pad the binary_cypher with the padded bytes. Finally, we get the cipher_text ready to be sent.

```
pad_removed_cipher = remove_pad_rsa(cipher_to_send)
# print(cipher_text)
decrypted_text = decrypt(pad_removed_cipher, private_key)
decrypted_message = digits_to_text(decrypted_text)

print(message, end="\n\n")
# print(parsed_message,  end="\n\n")
# print(cipher_text,  end="\n\n")
# print(decrypted_text,  end="\n\n")
print(decrypted_message,  end="\n\n")
```

Decryption process starts with removal of padded bytes and conversion of binary to ascii integers. The integers list is than decrypted and these digits are then converted to text.

Finally we print the messages.